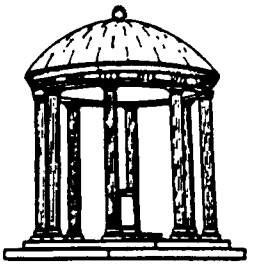
MoDE: A UIMS for Smalltalk

TR90-017
April, 1990
N00014-86-K-0680

Yen-Ping Shan



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A TextLab Report

MoDE: A UIMS for Smalltalk

Yen-Ping Shan

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
shan@cs.unc.edu
(919) 962-1874

Abstract

While the Model-View-Controller (MVC) framework has contributed to many aspects of user interface development in Smalltalk, interfaces produced with MVC often have highly coupled model, view, and controller classes. This coupling and the effort required to use MVC make user interface creation a less effective aspect of Smalltalk.

The Mode Development Environment (MoDE) is a user interface management system (UIMS) which addresses the above issues. MoDE is composed of two major components: the Mode framework and the Mode Composer. The Mode framework accommodates an orthogonal design which decouples the user interface components and increases their reusability. The Mode Composer reduces the effort of using MoDE by providing a direct-manipulation user interface to its users. This paper discusses the importance of orthogonality and illustrates its incorporation into the design of MoDE. A comparison of the Mode framework and the MVC framework is included.

1 Introduction

Smalltalk [2] has been a nice environment for developing experimental software. Its carefully designed programming environment and its rich class library allow exploring a design alternative in a short amount of time. The Model-View-Controller (MVC) framework [1, 3] is the major means of building user interfaces in Smalltalk. Although MVC framework has contributed to many aspects of user interface development in Smalltalk, it also has some shortcomings. While the MVC concept provides a compelling object-oriented division at the abstract level, concrete implementations often result in highly coupled model, view, and controller classes. Such coupling impedes the reuse and interchange of software components and at the same time produces awkward inheritance structures. Also, substantial learning effort is required before a programmer can effectively use MVC. Even for experienced MVC programmers, the time spent in creating a new user interface is still a substantial portion of the overall system development time.

The Mode Development Environment (MoDE) is a user interface management system (UIMS) that addresses the above issues. MoDE is composed of two major components: the Mode framework and the Mode Composer. Based on the high-level concepts of the MVC framework, the Mode framework emploies an orthogonal design to decouple the appearance, interaction and semantics components of an interaction technique. This not only allows better reuse of these components, but also results in a more flexible framework. The Mode Composer is the direct-manipulation interface of MoDE. Users of Mode Composer create interfaces by dragging objects out of the interaction technique library and pasting them together. With the Mode Composer, the effort and time required to create interfaces in Smalltalk is greatly reduced.

After a brief discussion of the Smalltalk MVC paradigm and its problems in the next section, Section 3 defines the concept of "mode." Section 4 illustrates why the orthogonality introduced in Section 3 supports generality and good reusability of user interface components. Section 5 introduces the kernel classes of the Mode framework. Section 6 compares the Mode framework with the original MVC framework. The Mode Composer is described in Section 7. Section 8 discusses the experience with MoDE. Section 9 then closes with some final remarks.

2 MVC and It's Problems

The MVC paradigm divides the responsibility for a user interface into three types of objects:

Model: The model represents the data structure of the application. It contains or has access to information to be displayed in its views.

View: The view handles all graphical tasks; it requests data from the model and displays the data. A view can contain subviews and be contained within superviews. The superview/subview hierarchy provides windowing behavior such as clipping and transformations.

Controller: The controller provides the interface between its associated model/view and the user input. The controller also schedules interactions with other controllers.

The three parts of a user interface are interconnected as shown in Figure 1. The standard interaction cycle is that the user takes some input action and the active controller responds by invoking the appropriate action in the

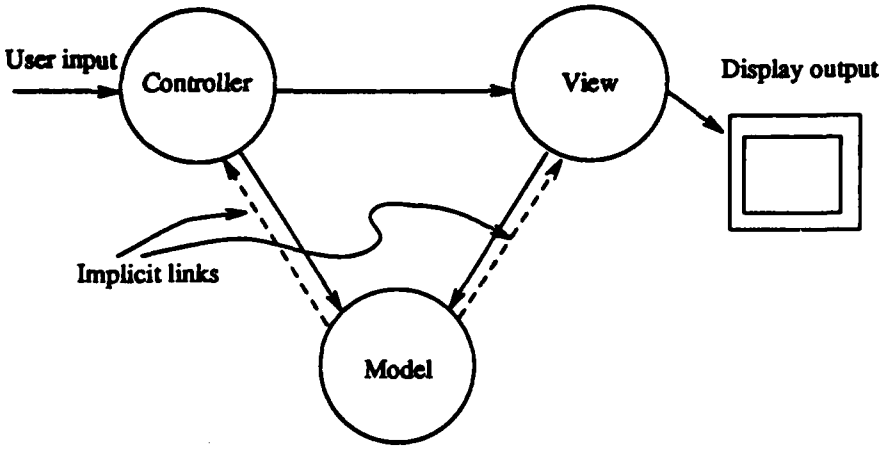


Figure 1: The Model-View-Controller framework.

model. The model carries out the prescribed operation, possibly changing its state, and broadcasts to all its dependent views (through the implicit links) that it has changed. Each view can then query the model for its new state and update its display, if necessary.

Although the MVC concept provides a convenient object-oriented division at the abstract level, the division is rather hard to implement. In Smalltalk, the MVC framework is implemented as three abstract superclasses (namely *Model*, *View*, and *Controller*). Numerous subclasses of the three abstract superclasses implement the interaction techniques used in the Smalltalk. Almost every model has a special view and controller pair associated with it. For example, the FillInTheBlank model has the FillInTheBlankView and the FillInTheBlankController. When this is done, the use of a controller, for instance, is limited to the view and model with which it is associated. Assigning a different controller to a view does not change the interaction but often breaks the code. As explained in Section 4, this kind of coupling often hinders the reuse of software components and produces awkward inheritance structures.

Although the MVC concept has its problems, its principle of dividing user interface components into three parts can still be used to guide the design of orthogonal interface components. While object-oriented inheritance alone does not guarantee good reuse of user interface components, an orthogonal design of those components, along with inheritance, can facilitate reusability. In addition, orthogonality results in a more general and versatile system for building user interfaces. The next section discusses the Mode framework that accommodates such an orthogonal design.

3 A Design for a Mode Framework

The basic building block of the Mode framework is called a *mode*. Each interface created with MoDE is composed of a number of such modes. A mode is distinguished by an area on the screen in which interactions with the user are different from those in its surrounding areas. A user interface might be composed of a group of hierarchically structured modes. A mode in such a structured interface could contain other modes as submodes. Any given mode, however, would be a submode of only one mode – its "supermode." The set of modes in a structured interface forms a hierarchy. The composition of modes in the Mode framework is analogous to the composition of views in MVC.

To illustrate, the dialogue box shown in Figure 2 can be thought of as a mode with two submodes: a "yes" submode and a "no" submode. The yes and no buttons (modes) highlight themselves when the left mouse button is pressed within them, and they dehighlight themselves when the cursor moves away or the left mouse button is released. Their behavior is different from that of their super-mode (the containing dialogue box) which does not respond to a left mouse button press. Notice that the text in the dialogue

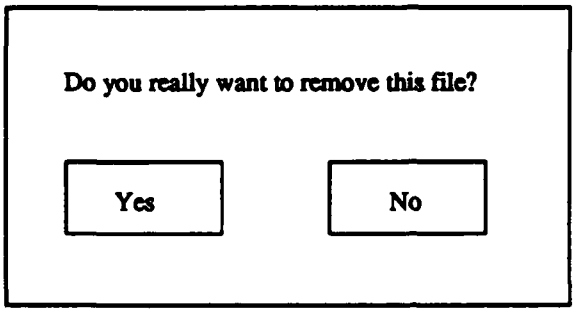


Figure 2: A dialogue box can be viewed as a mode with two submodes.

box is not a mode. It affects the appearance of the dialogue box, but it does not form an area that provides a different interpretation of the user's input.

Each individual mode is defined by its *appearance*, its *semantics*, and the form of *interaction* it provides. For example, the "yes" submode has the following definition:

Appearance: White background with black border of width one and a piece of text ("yes") centered. The highlighted appearance is the inverse of the normal appearance.

Semantics: Confirm to remove the file.

Interaction: Highlight when the left mouse button is pressed inside the mode; dehighlight when the cursor leaves or the button is released.

Notice that the "no" submode shares exactly the same interaction part with the "yes" submode. The differences between them come from the appearance and semantics parts.

In an object-oriented design, a mode is an object. The appearance, semantic, and interaction components are objects, as well. They can be possessed by mode objects, as shown in Figure 3. The mode object defines an

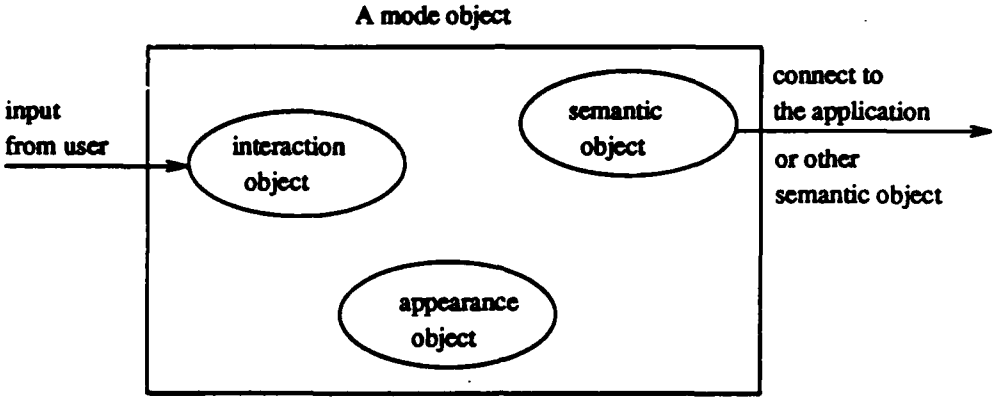


Figure 3: The structure of a mode.

internal protocol so that the component objects can communicate with each other in a standard way. Because the mode object provides a structure in which the three component objects can be plugged and unplugged, a mode's appearance, interaction, and semantics can be changed by replacing these component objects. For example, a mode that highlights has two different appearance objects: one for normal state, the other for highlighted state. When the mode highlights, it replaces the normal display object with the highlight display object. When it dehighlights, the normal display object is switched back.

4 A User Interface Component Space and Its Axes

In the above design, a mode is the composition of three parts: the appearance object, interaction object and semantic object. By assigning an axis to each part, we can define a three-dimensional type-space for modes, as shown in Figure 4. Each point in the space represents a different mode type. The

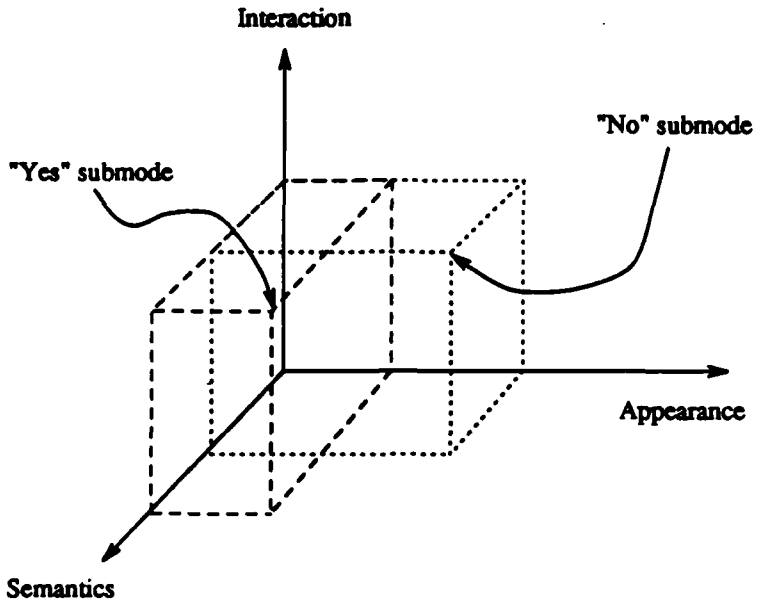


Figure 4: The three space for mode types. Two sample points are shown. One for the "yes" button, the other for the "no" button. They share the same interaction part.

"yes" and "no" submodes of the dialogue box example are shown as two points in the space. They have the same interactive behavior but different appearance and semantics. This is reflected in their sharing the same value on the "Interaction" axis.

Orthogonality of the Axes

Axes that span a space are orthogonal if changing the value on one axis does not affect the values on the other axes. That is to say, the axes are independent of one-another.

It is possible to represent the same mode-types with just one axis in which each type occupies a value on this single axis; however, this approach is less desirable since creating a new point on the axis defines only one new

type. In the case of a three-space, described above, creating a new point on one of the axes defines a plane of new types. In user interface construction, the one-dimensional approach represents lumping all three parts of a mode together in a single object. (Keeping them in three separate but closely coupled objects, like what has been done in MVC, is essentially the same.) The parts can only be reused when the whole object can be reused. In the three-dimensional case, three parts of a mode are put into three independent objects. The chances for each one of them to be reused are increased.

For example, assume an interaction technique library that contains two buttons. One is square-shaped and responds to a left mouse button click to perform operation A. The other one is round and responds to a middle mouse button click to perform operation B. In a single-dimensional design, to create a new button that is square-shaped and responds to middle button click to perform operation A, one would have to create a subclass of the first button and override the interaction. In a three-dimensional orthogonal design, such a button can be defined simply by replacing the interaction part of the first button with the interaction part of the second. No new class is needed. Actually, by permuting the parts, one can have 8 different buttons without creating any new classes.

This is a good example of how inheritance, alone, does not guarantee good reuse whereas an orthogonal design does. Notice that the three-dimensional orthogonal design is different from parameterizing the appearance and interaction of a single object. When a new appearance is invented (say a triangularly shaped display object), the three-dimensional approach immediately gives four additional new buttons. This is in contrast to the parameterized single dimension approach where editing the code and recompilation is necessary to incorporate the new shape.

Assuming the total number of types in the type-space to be N, a single

axis must have N distinct values to represent all the types. With three orthogonal axes, each axis would need only $\sqrt[3]{N}$ distinct values, in general. The number of distinct values for all three axes is $3 \times \sqrt[3]{N}$, as opposed to N in the single-axis case. In the above example, N is equal to 12, and the three-dimensional approach requires 7 values (three appearances, two interactions, and two semantics). The single-dimensional approach will need 12 values on its axis. Since N is usually much larger than 12, the three-space representation is also more efficient.

Generality

The generality of the user interface framework depends heavily on the choice of the axes. The more axes a framework has and the more orthogonal these axes are, the more mode-types it can span and the more general it is. In reality, it is difficult to find fully orthogonal axes. One can only strive for axes that are as orthogonal as possible. The Mode framework is an attempt to find one-such set of orthogonal axes as a demonstration of the concept. An implementation of this framework is described in the next section. New axes will evolve as new interaction techniques (for instance, sound) emerge.

5 Realization of the Mode Framework

The Mode framework is implemented on top of an event-driven mechanism [5] to avoid unnecessary performance loss and to provide a clean structure for interface programs. Four classes make up the Mode framework. They are *Mode*, *MController*, *MDisplayObject*, and *SemanticObject*.

The *Mode* class implements the basic structure of a mode. It has three instance variables to hold an *MController*, a *MDisplayObject* and a *SemanticObject*. A *Mode* coordinates the activities of these three objects to perform

the interaction. Besides that, a *Mode* is also responsible for handling various windowing functions (such as event dispatching and clipping). The *Mode* class also implements a simple constraint system to manage the layout of the user interface.

An *MController* performs the interaction by sending out messages according to the types of events it receives. The instance variable "eventResponses" of this class holds a dictionary that stores the mapping between the event types and the messages. Upon receiving an event, an *MController* tries to process it locally. When semantic actions are required, a message is sent to the semantic object to pass it the control.

A *MDisplayObject* defines the appearance of a mode by maintaining a collection of displayable objects. Any object that understands the protocols defined in the Smalltalk *DisplayObject* class can be put into the collection. This includes text, drawings, bitmaps, and animated pictures. A *MDisplayObject* accepts a display box and a collection of visible rectangles from its mode to display its contents.

A *SemanticObject* supplies the semantics of a mode. The term "supply" is used instead of "generate" because in MoDE, the actual semantics are "generated" by the application but they are "supplied" to the interface by a separate "semantic object," described here. Semantic objects can also connect to each other. They reside in a layer maintained by MoDE. Objects in the layer have knowledge of both the user interface and the application. They insulate both sides from the effects of changes.

The *MController*, *MDisplayObject*, and *SemanticObject* define parts of a mode that are approximately orthogonal to each other. As a consequence, they are more likely to be reused.

6 A Comparison to MVC

This section discusses some of the differences between the Mode framework and the Smalltalk MVC framework to show how the parts of a mode can be decoupled. The decoupling of the parts of a mode demonstrates the orthogonality of the Mode framework design. Although the comparison is made only for two specific systems, many of the points are applicable to more general cases. In the following discussions, the model, view, and controller of MVC are compared with their counterparts in MoDE.

Controllers

In MVC, controllers are often involved in processing the semantics in addition to their defined role as interface objects. For example, many controllers are responsible for creating menus, invoking them, and executing the selected operations. Many subclasses of *Controller* are created just to have different menus. For instance, the *IconController* and the *ProjectIconController* are the same except for their menus. In MoDE, controllers are not involved in semantic processing. They invoke menus to interact with the user but leave the creation of menus and execution of the operations to the semantic objects. Since the controller does not have deep knowledge of the menus, it is less tightly coupled to the semantics of the system. This reduces the number of controller classes while making the existing controllers more reusable.

A rough analogy can be drawn between a user interface and a restaurant. The controllers in a user interface correspond to the waiters in a restaurant. The semantic objects correspond to the cooks. In a normal restaurant, the cooks defines the menu and prepare whatever is on the menu. A waiter brings the menu to the customer (corresponding to the end user) and passes the selections back to the cooks. This procedure is analogous to the way

MoDE handles menus[1]. Just as there is no reason for the waiters to define the menu and cook the dishes, there is no reason for the controller to create the menu and perform the operations (as some MVC controllers do).

In MVC, some controllers (*BinaryChoiceController*, for example) query the state of their models to determine what kind of interaction to perform. This couples the controllers with their models. In MoDE, when the state of a semantic object changes and requires a different interaction, a different controller is assigned to the mode. No controller should have to query the state of its semantic object. This approach is actually used in MoDE to provide semantic feedback for dragging. When a mode is dragged by the user, all other modes on the screen switch to their drag-handling controllers. For example, the trash mode switches to a controller that highlights the mode when the dragged object is on top of it and responds to the mouse button release event to discard the dragged mode. The trash mode switches back to its normal controller after the drag action is finished.

Another limitation on MVC controllers which impedes orthogonality is their polling protocol. The MVC controllers must constantly query their views for the information necessary to decide when and where to pass control. The event-driven mechanism of MoDE takes charge of the control passing. This frees the controller from querying the mode and makes the two less dependent on each other.

Views

Some MVC views also overstep their authority by incorporating semantic information. These views often keep information and code that could be decomposed and distributed more appropriately among semantic objects and subviews. For example, the *SelectionInListView* keeps the list of items, remembers which one of them is selected, and highlights or dehighlights the

[1]Some Smalltalk pluggable views handle their menus similarly.

items. The *SelectionInListView* has to do all this because it is at the end of the view hierarchy (it has no subviews). The list items are not subviews.

With the Mode framework, on the other hand, each list item is a mode and knows how to highlight and dehighlight itself. The instance variables and the code to handle the selection are moved to their semantic objects. This not only simplifies the interface but also makes it more flexible. For example, one can use bitmaps, drawings, and animated pictures in the display object of the list item modes to create a nontext list. One can also freely select the highlight styles for each individual list item (as opposed to having a single fixed inverse highlight for all of them).

Smalltalk menus, which are not built with MVC, provide a related example. A Smalltalk menu is a single complicated object. In MoDE, menus are built with modes: each menu item is a mode; this makes the menus more flexible. Item modes can also share components with the list mode.

Models

In MVC, models do not have direct access to their views and controllers. When a model changes, a message is broadcast to notify all of its views and controllers. The views and the controllers then query the model and update themselves to reflect the change. This has several disadvantages. For example, the model may be a widely shared data object that has a large number of views. Having all the views query it whenever there is a change is costly. Also, the broadcast mechanism usually requires smart user interfaces that know how to query the models and update themselves. The code that supports this intelligence goes to either the view class or the controller class. Thus, knowledge of the application (model) is inserted into the user interface. Once this is done, the model, view, and controller are, in fact, coupled.

The Mode framework solves this problem by abstracting this intelligence

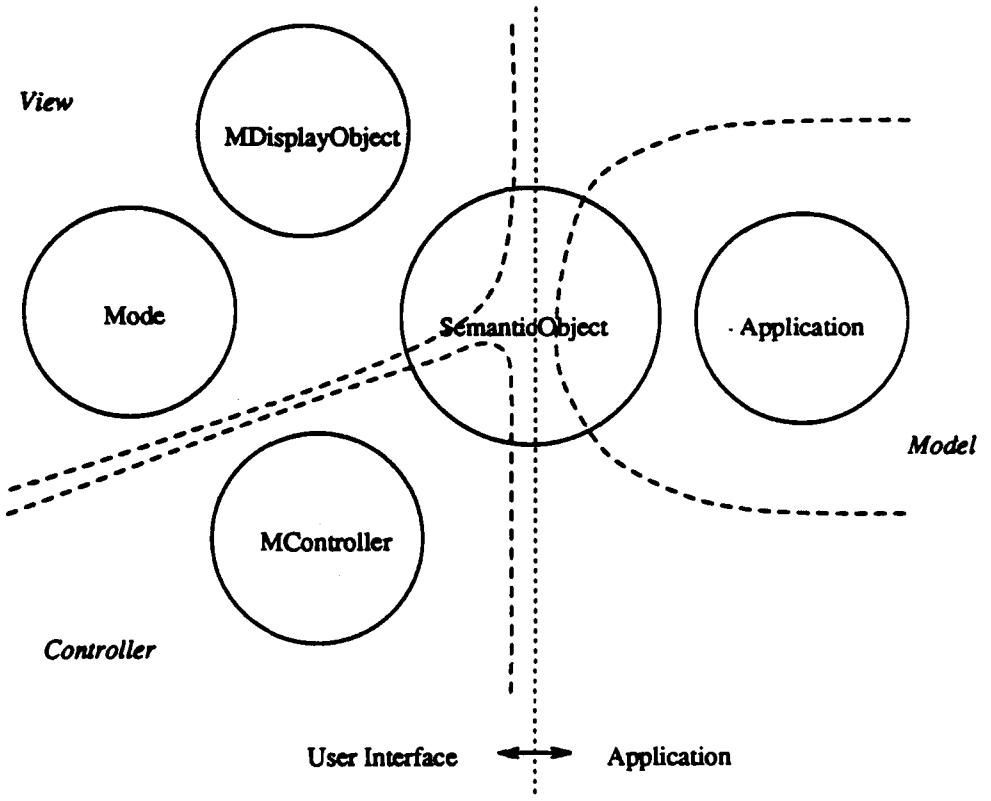


Figure 5: The responsibilities are partitioned differently in the Mode framework than in the MVC framework.

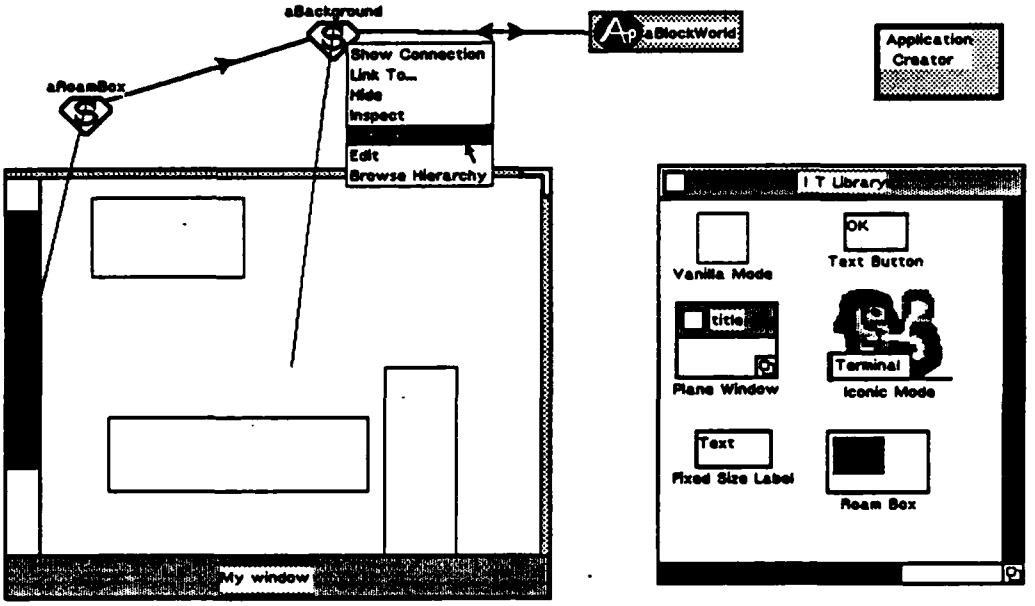


Figure 6: Using the Mode Composer.

into the semantic object. This frees the other objects from the need to be coupled with each other. Figure 5 shows the partition of responsibilities in the Mode framework and in the MVC framework. The circles indicate the objects in the Mode framework. The dashed lines show the corresponding MVC objects (their names are in italics).

7 Mode Composer

The Mode Composer is the direct-manipulation user interface of MoDE. It allows the user to create an interface, edit it, and connect the interface to the application through direct manipulation.

The user creates interfaces by dragging modes out of the interaction tech-

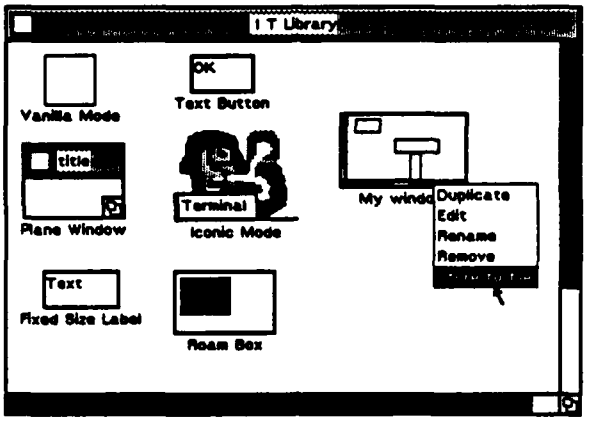


Figure 7: The interaction technique library.

nique library (the right-hand window in Figure 6) and pasting them together. Visual representatives of semantic objects and application objects can be created and manipulated directly. In Figure 6, the user has finished the layout and connection of the interface (an upside-down window) and is asking the system to create a subclass of the "aBackground" semantic object.

All interfaces created with the Mode Composer are immediately testable at any stage of the development. Thus, there is no need to switch to a test state. After the interface is created and tested, it can be promoted into the library for future use. In figure 7, the upside-down window has been promoted into the interaction technique library and is represented by an icon. The user can also store it in a file and share it with other user interface developers.

The interaction technique library of the Mode Composer stores prototypes [4] of the interaction techniques. As a consequence, each library object represents a prototype instead of a class. Once an interaction technique is promoted into the library, it can be reused immediately by making copies of its prototype.

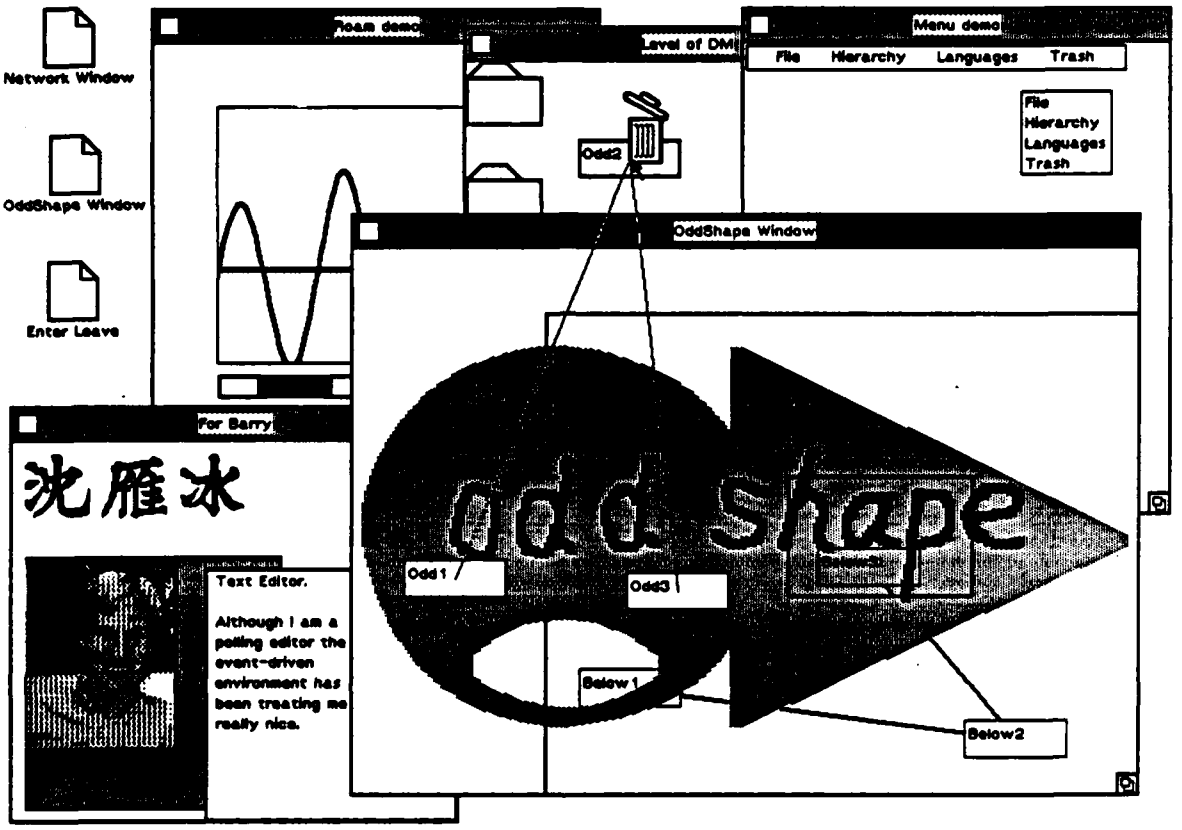


Figure 8: Sample user interfaces created with MoDE.

Besides the orthogonal design of the Mode framework, the capability to easily introduce new objects into the library is also essential to the generality of the system. If an interface builder were to have a fixed set of library objects, the kinds of interfaces that it could create would be limited. Since the user of MoDE can freely promote new objects into the interaction technique library, MoDE is not limited in this respect.

8 Experience with MoDE

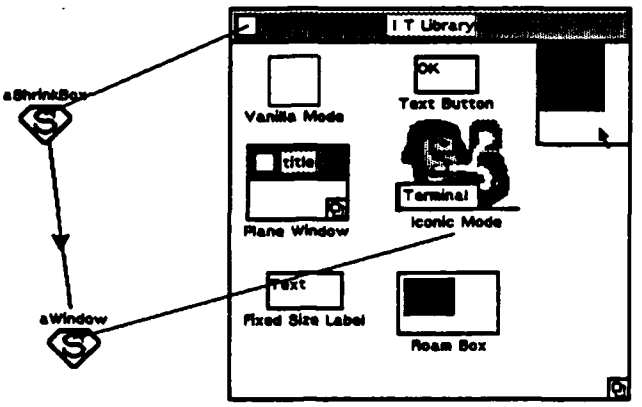


Figure 9: The Mode Composer is used to edit itself.

Sample Interfaces

MoDE has been used to create many direct-manipulation user interfaces. Figure 8 shows a few sample interfaces created with it. The scroll bar in the top left window (Roam demo) scrolls the picture continuously. The top right window (Menu demo) has three types of menus: title-bar menu, tear-off menu, and pop-up menu (not displayed). Menu items can be text, foreign characters, bitmaps, and animated pictures. The lower left window (titled "For Barry") demonstrates the system's capability to incorporate scanned images and text editors. The largest window (titled "OddShape Window") contains two subwindows; both allow the user to create networks of hypertext nodes. The oddly shaped subwindow has three nodes in it. The user is dragging one of the nodes over the trash icon in another window (titled "Level of DM"). The trash icon opens to provide semantic feedback. Rubber-band lines are drawn from "Odd1" node and "Odd3" node to the node being dragged to show the connection. Notice that the oddly shaped subwindow has a hole in it through which the user can work with objects (for example, the "Below1" node) underneath the window. MoDE also supports semi-transparent windows as shown in the right half of the oddly shaped subwindow.

Self-Creation

Not only is the Mode Composer an important component of MoDE, it is also an important application of MoDE. To demonstrate the generality of MoDE, the user interface of the Mode Composer was created using itself. Consequently, the Mode Composer can be used to edit itself. For example, in Figure 9, the user is using the Mode Composer to examine the connection between the "ShrinkBox" and the "Window" of the interaction technique library. The user has also made some changes to the Mode Composer. The two scroll bars of the interaction technique library have been removed, and a "Roam Box" (a two-dimensional scrolling device) has been attached.

Since it is easy for users to customize the user interface of the Mode Composer, the interface images shown in this document represent only a small sample of those developed by the author.

Effectiveness

In an informal experiment to study the effectiveness of MoDE, two groups of subjects were asked to create the same problem interface. One group (consists of Smalltalk programmers with less than three months experience) used MoDE exclusively; the other group (consists of Smalltalk programmers with 1.5 to 5 years experience) used whatever tools they liked except MoDE. The group using MoDE were able to finish the assignment both faster and with fewer unimplemented features than the other group. Time data collected from this informal experiment suggest that MoDE reduces the time required to develop a prototype interface by nearly an order of magnitude.

Classes Do Not Make Good Types

The interaction technique library is an interesting example of classes not making good types. Observation shows that the users of MoDE naturally treat each object in the library as a type. For example, a user might drag a button out of the library, change its border width, and promote the changed

button back to the library. From then on, he would think he has two types of buttons instead of one. The same thing happened to changes made to the controller and the semantic object. Even though the two buttons are composed of parts from the same classes, they are treated as different types. Since differences in the interaction technique library come more from the values of the instance variables of the objects than the classes to which they belong, classes are not sufficient to differentiate these types. This supports the choice of using prototypes which preserve the values of the instance variables, instead of classes, to represent objects in the interaction technique library.

9 Conclusion

MoDE provides an effective environment for developing user interfaces in Smalltalk. The capability to uncouple the parts of a mode not only increases reusability but also results in a more flexible system with which a wide variety of interfaces can be developed. Experience with the Mode Composer indicates that its direct-manipulation interface substantially reduces the time and effort required to create and manage user interfaces in Smalltalk.

MoDE is currently being used to create interfaces for a hypertext software development system. The author is also exploring the possibility of augmenting MoDE to automatically generate code for X Window System. (A similar project has been done by hand.)

10 Acknowledgement

A number of organizations and people have contributed to the work reported here. The author is grateful to the National Science Foundation (Grant # IRI-85-19517) and the Army Research Institute (Contract #MDA903-86-C-0345) for their support of this research. This work has been done as part of the author's dissertation project under the supervision of Professor John B. Smith. Barry Elledge and John Maloney provided valuable comments and suggestions for this paper. The Textlab Research Group within the Department of Computer Science at the University of North Carolina at Chapel Hill has provided a provocative and supportive intellectual environment for this work.

References

[1] Sam S. Adams. Metamethods: The mvc paradigm. *HOOPLA!*, 1(4), July 1988.

[2] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[3] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[4] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *OOPSLA'86: Object Oriented Programming, Systems and Applications*, pages 214–223, September 1986.

[5] Yen-Ping Shan. An event-driven model-view-controller framework for smalltalk. In *OOPSLA'89: Object Oriented Programming, Systems and Applications*, pages 347–352, October 1989.